SYSTEM AND METHOD FOR MULTI-PLATFORM QUEUE QUERIES

Cross-Reference to Related Application

This patent application is related to and claims the benefit of Provisional U.S. Patent Application No. 60/459,328 filed on April 1, 2003, which application is hereby incorporated herein by reference in its entirety.

Field of the Invention

The present invention relates to the field of distributed data processing, and, more specifically, to a system and method that permits a user to determine length and content of process queues across multiple platforms in a distributed data processing system.

Background of the Invention

In modern distributed data processing systems, developers and users alike frequently have time-sensitive communications needs. For example, a developer needs to know how to route messages so that they are expeditiously delivered. A user may also need to know what is happening to a time-sensitive transaction. In both cases, the user needs to know what is happening with messages on communication queues.

Most systems have some queue query function. These functions, however, frequently can only query one queue at a time and must be run repeatedly to determine a complete picture of the state of the system. Other systems can only query a specific platform and not the entire distributed system. Still others require that the user have administrator privileges on one or more of the platforms in the distributed system.

Therefore, a problem in the art is that there is no manner in which a developer or user can obtain complete information about the state of queues in a distributed system.

Summary of the Invention

This problem is solved and a technical advance is achieved in the art by a system and method that provides a cross-platform queue viewer for use in a distributed processing system comprising a plurality of operational platforms that cooperate to perform various

functions and tasks. The queues may be, for example, message queues in a distributed operating environment (such as JAVA).

According to an exemplary embodiment of this invention, a web browser is in communication with an application server. The web browser provides the application server with information regarding a query request. The application server communicates this request to one or more message servers. The message server communicates queue information responsive to the query back to the application server. The application server processes this information into a form that is easily understood by the requester.

Advantageously, the application server comprises a J2EE application server, which sorts queue information received into a plurality of categories. A tree renderer advantageously receives the sorted categories and derives a tree structure, which is delivered to the web browser.

Brief Description of the Drawings

A more complete understanding of the invention can be obtained from a consideration of the specification in conjunction with the drawings, in which:

FIG. 1 depicts a block diagram of an application architecture for an illustrative embodiment of the present invention;

FIG. 2 depicts a block diagram of the remote method invocation runtime structure for message communication of FIG. 1;

FIG. 3 depicts an exemplary UML class diagram of the illustrative embodiment of the present invention;

FIG. 4 depicts a UML use case diagram of the illustrative embodiment of the present invention;

FIG. 5 depicts a first screen shot of an exemplary invocation of this invention;

FIG. 6 depicts a second screen shot of an exemplary invocation of this invention; and

FIG. 7 depicts a third screen shot of an exemplary invocation of this invention, illustrating the results of a multi-platform queue inspection.

Detailed Description

This invention introduces a new capability into the art: the ability to query multiple queues on multiple, distributed platforms and then report the results in an easy-to-understand format. For purposes of this application, this capability is called "Message Inspector." In order to aid in the understanding of the functionality of this exemplary embodiment of this invention, the goals of this invention are presented.

1. Easy to Use

As most people are familiar with using Microsoft Explorer's tree-like navigation interface, this exemplary embodiment of this invention takes advantage of a generic tree navigation interface available as a common utility (within the GOODS toolkit), which provides for reuse.

2. Simple to Set Up

This exemplary embodiment of this invention employs Java Server Pages (JPS) and Java Beans for the front end running on a Websphere application server. This implementation has the advantage of no client-side installation, which enables a user to access Message Inspector from any machine with a web browser. On the Message Server side (as will be described further, below), a Java RMI server is installed, which carries out Java Message Service (JMS) work using JMS Provider Java. Therefore, there is a once-only installation on the Websphere application server and a once-only installation on each JMS Provider server machine.

3. Efficient

The user interface on the Websphere application server must be responsive (like any browser application). To this end, Message Inspector includes a user-configurable message limit, which defaults to 50 messages. If a queue is filled to its respective capacity, then it is not necessary to retrieve all messages and wait for the tree to be redrawn. However, if the user does want to see all messages on a queue, he or she is able to do so by choosing to set the message limit to the maximum.

4. Robust

The idea behind Message Inspector is to aid production support and application development and not to add to their work load. Thus, Message Inspector is a robust application which requires little or no intervention once it is installed. Error handling is given a high priority and all problems are reported to its own error logs. Message Inspector is started when the underlying platform is rebooted and the J2EE Application Server (as will be described below) is running.

5. Read Only

Message Inspector provides a read-only interface so no one can inadvertently remove messages or queues and affect the JMS Provider's guaranteed messaging environment.

6. Secure

Messages may contain sensitive business data that only authorized personnel should be able to access with Message Inspector. This authorization is achieved by using a web portal login servlet which authenticates a user against a valid database account.

FIG. 1 shows the application architecture 100 of an illustrative embodiment of a Message Inspector according to this invention. The three main components illustrated are the Websphere Application server 102, a J2EE Application Server 104 and a JMS Messaging server 106. There may be more than one JMS Messaging server, as indicated by 108.

A web browser 110, as is well known in the art and therefore not further discussed, is running on the Websphere Application server 102, which is illustrated as running Message Inspector. A tree structure 112 is illustrated in web browser 110. "+" signs indicate that there are further entries. The user may expand the visible tree structure 112 by activating (for example, by way of a mouse click) a "+" sign. Tree structure 112 is delivered to Websphere application server 102 and web browser 110 from J2EE Application Server by means of HTML (which is well known in the art and therefore not further discussed).

The J2EE Application Server 104 components added to implement the illustrative embodiment of this invention are shown in box 104. The GOODS HTML Tree Renderer 114 generates HTML tree structure 112. A Message Inspector tree 116 includes the following nodes or categories: the Queues 118, the Queue Managers 120, the Messages 122 and the Messaging Servers 124. Each category is represented by a bean, which generate input to Tree

Renderer 114 and is responsible for generating the list of entities (*i.e.*, queues, messages, *etc.*) at a position in the tree.

Also included in the J2EE Application Server 104 is a Message Inspector RMI Client 126. Message Inspector RMI Client 126 is responsible for all RMI interaction with a Message Inspector RMI Server 128 within JMS Messaging Server 106. Message Inspector RMI Server 128 acts as a proxy for calls to the JMS Messaging Server 128. The JMS Messaging Server 106 represents the server machine which actually holds queue managers and their respective queues.

In operation, Message Inspector RMI Client 128 is controlled by the set of java beans (categories) 118, 120, 122 and 124, which act as data sources for the HTML Tree Renderer 114. The Tree Renderer 114 generates HTML to send to the web browser 110. The list of JMS Messaging Servers 106, 108 is held in a properties file in Message Inspector RMI Client 126, as is the RMI registry number. Addition of a new JMS Messaging Server, such as 108, requires a restart of the application. Everything else is found at runtime.

The set of java beans (categories) 118, 120, 122 and 124 use RMI communication when they require interaction with JMS Messaging Server 106. Each JMS Provider's Messaging Server 106, 108 includes an installed Message Inspector Server 128. This is an activatable RMI Server (*see*: http://java.sun.com/products/jdk/rmi). The Message Inspector RMI Server 128 is responsible for all interaction with the JMS Messaging Server 106 and uses the JMS API. The Message Inspector RMI Server 128 extends the "java.rmi.activation.Activatable" class, which allows it to be invoked remotely, provided it is registered with the RMI activation demon.

A setup class is also provided, which runs on the JMS Messaging Server 106. This class, called "SetupMessageInspector," declares the Message Inspector RMI Server's 128 remote interface and registers it with the RMI activation demon, binding the stub to a name in the RMI registry. This needs to be run when the JMS Messaging Server 106 is restarted. A script is provided for this purpose that runs the RMI registry, the RMI activation demon and the SetupMessageInspector class.

Turning now to FIG. 2, a brief illustration of RMI processing is shown. FIG. 2 shows the RMI Runtime structure used to make a call to the method "getMessages." A top layer is

the application specific layer 200, which includes the Message Inspector RMI Client 126 and the Message Inspector RMI Server 128. The next layer is the RMI layer 202 wherein the RMI Stub 204 implements the interface methods in the application client simply by relaying the method invocation 206 to the RMI Skeleton 208 on the server side. The method invocation 206 is relayed across the network layer 210 via the RMI Runtime services 212 on the Application Server 104 over a network 214 to the RMI Runtime services 216 on the Messaging Server 106. One skilled in the art will appreciate how to construct the RMI structure after reviewing this specification and the article at: http://java.sun.com/marketing/collateral/rmi_ds.html, which is incorporated herein by reference in its entirety.

FIG. 3 illustrates a UML class diagram 300 for Message Inspector. Tree structure 112 is rendered by Tree Renderer 114, which uses TreeNodeExtractor 302 objects as containers for the data. Each of the Bean classes (*i.e.*, Queue Manager Bean 120, Server Bean 124, Queues Bean 118 and Messages Bean 122) inherits from TreeNodeExtractor 302. The Servers Bean 124 contains the list of servers that have the Message Inspector RMI Server 128 installation. The Queue Manger Bean 120 contains the list of queue managers on a particular server. The Queue Bean 118 contains the list of message queues on a particular queue manager. The Message Bean 122 contains the list of messages on a particular queue. The Message Inspector RMI Client 126 managers the interaction with the Message Inspector RMI server 128. Message Inspector RMI Server 128 supplies the beans with the data they require. Message Inspector RMI Server 128 and Message Inspector RMI Client 126 each implement the required RMI interfaces: RMI Server 310 and RMI Client 312, respectively.

FIG. 4 illustrates the set of use case scenarios for a message inspector application according to an exemplary embodiment of this invention. FIG. 4 uses standard UML notation after Jacobson (1994). In order to illustrate more clearly the interaction of each component, each use case scenario is examined in turn. Note that where the term "Message Inspector" is used it is implicit that this is Message Inspector RMI Client 126 and Message Inspector RMI Server acting together as one object. FIG.'s 5-7 are used herein as specific examples of what a user might expect to be displayed on web browser 110.

1. Select Server

a. User 400 selects server 402, see screen shot FIG. 5.

b. Java script on web browser 110 calls "onclick" method and the server name is sent to the web server object, Tree Renderer 114.

c. Tree Renderer 114 calls Queue Manager Bean 120.

d. Queue Manager Bean 120 calls Message Inspector RMI Client 126 "setServer" method, which then invokes connect to server scenario, 2.

e. A list of queue managers is returned to the Queue Manager Bean 120 and the Tree Renderer 114 then generates the HTML to return to the web browser 110, see screenshot FIG. 6.

2. Connect To Server

a. Message Inspector RMI Client 126 creates a connection to the Message Inspector RMI Server 128 in 404.

b. If this fails then an error message string is returned to the Queue Manager Bean 120 in 405.

3. Get List of Queue Managers

a. Message Inspector RMI Server 128 searches the JMS Messaging Server for installed queue managers in 406.

b. Message Inspector RMI Server 128 returns the list of queue managers to the Message Inspector RMI Client 126.

4. Select Queue Manager

a. The user selects a queue manager in 408 and the queue manager name is sent to Tree Renderer 114.

b. Tree Renderer 114 passes the queue manager name to the Queue Bean 118.

c. Queue Bean 118 calls Message Inspector RMI Client 126 "setQueueManager" method.

d. Queue Bean 118 then calls the Message Inspector RMI Client 126 "getQueues" method.

5. Get List of Queues

a. The Message Inspector connects to the queue manager in 410 and gets a list of queues in 412.

b. This list of queues is returned to the Queue Bean 118.

c. For each queue, the Message Inspector then gets the count of messages on that queue (the queue depths) in 414.

d. Tree Renderer 114 uses the Queue Bean 118 to generate HTML for the web browser 110 showing the list of queues and the number of messages on each queue, see FIG. 7.

6. Connect to Queue Manager

a. The Message Inspector gets a connection to the queue manager 410.

b. If this fails then an error message string is returned to the Queue Bean 118 in 416.

c. This error message is displayed by the Tree Renderer 114 in an HTML page on web browser 110.

7. Get Messages

a. The Message Inspector selects a queue in 418 and retrieves the messages on the currently selected queue 420.

b. These messages are returned to the Message Bean 122 and the Tree Renderer 114 uses this bean to generate an HTML page for web browser 110 showing the messages.

8. Display Message

a. When a message is selected in 422, a java script is called to show the message in a popup window in 424.

b. If the message is XML then the message is displayed in an XML format. If the message is not in XML, then the message data is just displayed as is.

It is to be understood that the above-described embodiment is merely illustrative of the present invention and that many variations of the above-described embodiment can be devised by one skilled in the art without departing from the scope of the invention. It is therefore intended that such variations be included within the scope of the following claims and their equivalents.